

## **8. DCE-Based Applications**

The DII COE is designed to support applications using the distributed client/server computing model. There are many ways to implement a distributed client/server environment. The DII COE provides the Open Software Foundation's (OSF) Distributed Computing Environment (DCE) as a baseline for distributed architecture/standards. To be DII-compliant, there is no requirement to use DCE as the baseline for a client/server implementation, or that segments be client/server-based. However, if the application uses RPCs (Remote Procedure Calls), they must be compatible with DCE RPCs.

DCE is an integrated set of services that support the development, use, and maintenance of distributed applications. A set of written standards and a package of developer's software are available from the OSF.<sup>1</sup> Based on these, a large number of applications have been written by various software vendors for end users. Use of DCE is not restricted to Unix environments. Clients or servers may operate on other operating systems, although the most applications employ Microsoft Windows or Windows NT clients and Unix servers.

The purpose of this chapter is to provide the minimum essential information necessary for DCE COE and mission-application developers to begin developing DCE mission applications. It is not a tutorial on DCE, nor does it provide an in-depth discussion of development tools, management procedures, or compliance criteria (in the sense of DCE standards). Developers using DCE should refer to OSF or vendor documentation for general guidance on DCE.

The DII COE provides a COTS reference implementation of a DCE server and a DCE client. Developers shall use these rather than providing their own copy of an alternative COTS DCE product. This is required of all segment developers, including mission-application developers, because the end COE-based system is likely to be installed on a LAN that includes multiple COE-based systems.

---

<sup>1</sup> DISA maintains a facility called the Operational Support Facility in the Washington, DC area. Throughout this chapter, unless otherwise indicated, OSF refers to the Open Software Foundation and *not* to DISA's Operational Support Facility.

## **8.1 DCE Overview**

OSF's DCE is commercial software that provides a comprehensive set of services that support the development, use, and maintenance of distributed applications. DCE allows diverse systems to work together cooperatively and masks the technical complexities of the network. Because DCE is independent of the operating system and network, it is compatible with many diverse environments.

The strength and appeal of DCE stem from its ability to make a group of loosely connected systems appear as a single system to Information Systems (IS) staff, end-users, system administrators, and application developers. Applications executed under DCE take advantage of untapped resources on networks by finding the platform best suited for a particular job. Similarly, complex tasks can be easily split among multiple computers on the network to reduce computing time and improve performance. From a security perspective, users in a DCE-enabled computing network need only log in once for access to all network platforms.

Many compare the OSF's DCE to wiring or plumbing because it provides the underlying transport layer that enables distributed client/server applications to interoperate across a heterogeneous environment. DCE currently consists of the following services:

- Remote Procedure Calls (RPC)
- Cell Directory Service (CDS)
- Distributed Time Service (DTS)
- Distributed File Service (DFS)
- Security Service
- Threads.

### **8.1.1 Remote Procedure Call (RPC)**

The key to making many disparate resources function logically as one system within DCE is the RPC. In DCE, RPCs let multiple computers execute applications, or parts of applications, on the platform chosen by the developer as best suited for the task.

The RPC makes a wide variety of application capabilities possible that were previously either impossible or extremely difficult to implement. These capabilities include the following:

1. allowing multiple clients (in a client/server network) to interact with multiple servers, and multiple servers to handle multiple clients simultaneously,
2. the ability for clients, through DCE's Directory Services, to identify and locate network users by logical service name,
3. protocol independence across the network for any platform, and

4. secure communications across the network.

### **8.1.2 Cell Directory Services (CDS)**

The DCE Cell Directory Service (CDS) provides a single naming model throughout a distributed environment. Directory Services let users access network services, such as printers, servers, and other network platforms, by name, without the necessity of knowing where the resource is located within the network. This lets users access a network resource even if the resource has been moved to a different physical network address.

The Cell Directory Service can make use of its built-in X.500 Global Directory Service (GDS) for locating resources in external cells, or can make use of Domain Name Service (DNS) for this purpose. Cell names are constructed differently depending on which approach is selected.

- The DII COE will use DNS to locate external cells, and therefore will use DNS-style cell names.

### **8.1.3 Distributed Time Service (DTS)**

DCE DTS allows multiple platforms to work together to share information without timing problems that might affect event scheduling and duration. DTS regulates system clocks on each network computer so that they match each other. Clocks are synchronized, and the service ignores faulty system clocks. The DCE Time Service uses authenticated DCE RPC so that, unlike the Internet Network Time Protocol, the DCE global clock synchronization is secure. Also, to support network sites that wish to use time values from outside sources, DTS supports the Network Time Protocol standard. The DCE Time Service also includes a published Time Provider Interface to allow it to receive inputs from other reliable time sources, such as Global Positioning Satellite (GPS) or other military systems.

- DCE DTS provides intra-cell clock synchronization in the DII COE. Inter-cell synchronization is not supported.

### **8.1.4 Distributed File Services (DFS)**

The DCE DFS is a fundamental element for information sharing in DCE-enabled networks. It is one of many facilities that could theoretically be built on the foundation provided by DCE's Core Services. DFS unites the file systems of all network nodes for a consistent interface, making global file access as easy as local access. It replicates files and directories on multiple network machines for fast and reliable access, even when communication lines and network hardware fail. It also caches copies of currently used files at the requesting node to minimize network traffic and provide fast data access.

- DFS is not provided as part of the DII COE. Specific communities may implement DFS on top of the DII COE. Information in this chapter about DFS describes it as it is

planned to be used by the GCCS community. This may serve as a useful model for other mission domains.

### **8.1.5 Security**

While security maintenance and administration are simplified for one central system behind a glass wall, security for dozens of computers scattered across a wide area network, all operating as a single entity, is much more complicated. DCE's Security Services ensures distributed security. The Security Service software layer is made up of three parts: authentication, authorization, and user registry. DCE facilitates these services through the RPC, which maintains the integrity of information passed across the network.

The authorization mechanism grants authorized users access to resources and rejects requests from unauthorized users. DCE implements Access Control Lists (ACL) based on a draft Portable Operating System Interface for Unix (POSIX) standard that provides a fine-grained object/operation security authorization model.

The user registry permits users to access multiple network resources through a single password and single login. The registry is a single database of user information that is replicated around the network. User passwords and security-related attributes are centrally stored and universally available.

Many security features, including auditing, delegation, and a registry extension to support non-Unix systems, are provided by DCE. Improved security is one of the primary motivations for the movement to DCE for DII applications. OSF DCE provides the following significant features related to security:

1. DCE Authentication provides a secure mechanism (unforgeable) for establishing identity. A user should not be able to compromise the authentication process by using a 'root' account on any machine to project Unix credentials.
2. Authorization for execution of applications is based on DCE credentials in addition to Unix credentials. The granularity of execution control on a base Unix system is limited to an owner/group/world model that is not sufficiently flexible. As a result, almost all applications are set to enable world execute permission.
3. Authorization for operation invocation is based on DCE credentials. Most existing applications either do not have granular access decisions or have implemented their own means of access control. An example of the latter is a database server that may define roles as a means of protecting classes of operations. New applications and those being migrated should be provided with a consistent means of defining, managing, and performing these operations.

4. DCE security allows a client to securely project its identity, including memberships, in other security groups. This allows authorizations to be group-based rather than user-based.
5. Single-login allows all related access decisions to be based on the same distributed identity. Without this capability, users may be required to login to multiple systems or applications, and security administrators must keep multiple identities and security files in synchronization.
6. Execution auditing records DCE and Unix credentials. This records the identity of anyone running an audited application (see below).
7. Protection against packet insertion/replay, packet interjection, and eavesdropping can be achieved when using DCE RPCs at the appropriate security level or when using the Generic Security Services API (GSSAPI) to protect data transmitted over the network.

**Note:** For the near term, security for DII distributed applications will be provided by the DCE Security Service, which is based on Kerberos. The OSF and DOD are exploring ways to link DCE security with DOD techniques such as MISSI. Other security mechanisms may be provided in future versions of the DII COE as the COE migrates from a software-based security solution to a hardware-based solution.

### 8.1.6 Threads

The underlying Threads Service is used by several DCE services, including the RPC. Threads are programs that use “lightweight” processes to perform many actions concurrently. Threads are particularly useful in allowing server applications to process multiple requests concurrently. DCE Threads are based on the POSIX threads standard. OSF has designed the multi-threading capability of the Threads Service to be easily accessible by programmers wishing to use it in applications. Most commercial applications using threads are written in C, so these DCE services can be accessed through the C programming language. Bindings exist for Ada, as well as other high-level programming languages.

### 8.1.7 Client/Server Concepts

DCE is specifically designed to manage the distribution of processing across multiple platforms. It is a powerful infrastructure for building client/server architectures. The client/server computing model for DCE introduces a few additional terms.

1. In the DCE context, a *server* is a single executable program that provides services to clients. An example of a server is a DBMS, or a map server that provides map images to a calling application. A site can employ multiple servers to create a more available

or more balanced service environment. A DII segment can contain multiple servers each performing some related service.

2. A server implements one or more *services*, each of which is offered through an *interface*. Interfaces are well defined, using the DCE Interface Definition Language<sup>2</sup> (IDL), and are the concrete descriptions of a service. Usually, a server implements at least two interfaces. One provides the operational interface for client requests. The other provides a management interface (e.g., for security). Internally, all DCE servers implement other interfaces used for querying, stopping, or reconfiguring the server.
3. An interface provides access to one or more *operations*, each of which corresponds to a specific function or procedure call. For example, a complex math interface could provide separate operations for complex addition, subtraction, multiplication, and division. The operations within an interface should be very closely related.
4. In DCE, clients locate appropriate services by using the DCE CDS to provide the location of one or more servers. The client presents a CDS name (or listing) and optionally, a resource element (object UUID). The CDS name corresponds with the logical service name rather than a machine or hostname. This indirection allows DCE to provide location independence and employ multiple compatible servers for availability or load balancing.
5. Each operator using DCE is identified with a unique DCE *principal*. A DCE principal has a DCE account maintaining its DCE identifier (UUID) along with its Unix identity (uid, gid). A DCE principal will map uniquely to a Unix userid.
6. Each DCE server is also identified with a particular principal. For security reasons, server principals should map to Unix userids that are not allowed to login (i.e., without a login password). These Unix userids correspond to the concept of a “system account” (like uucp).
7. Although it is not necessary for the client and server to be installed on separate machines, one of the primary reasons for constructing client/server applications is to share access to a one or more server resource among multiple clients. Since the segment is the smallest installation unit, the client and server portions of an application are usually delivered in separate segments.

---

<sup>2</sup> The DCE IDL should not be confused with the CORBA IDL. Both are similar in concept, but differ in implementation.

## **8.2 DII COE DCE Services**

The DII COE supplements the COTS DCE product with a number of tools to assist the developer in creating segments that use DCE, and in installing and managing DCE at an operational site. Commercial products are preferable, but many of the tools and features required are not available commercially. The tools discussed in this section, and the DCE-related tools described in Appendix C, are specifically designed for the DII COE rules for DCE applications. In addition, development of DCE guidance for the COE highlighted some issues/items that must be addressed in order to assist in the development of DCE mission-application segments, and implementation of DCE in the COE.

### **8.2.1 Standard Server Installation**

The first part of a DCE server installation process must run as root. Installation of the DCE server has been standardized for the COE and is part of the DCE COE-component segment. Installation uses a parameterized `dcecp` script to create an initial CDS entry and principal for the segment, and give it permissions to create the rest of the structure.

### **8.2.2 Standard Server Initialization**

A secure DCE server must make between 7 and 30 DCE calls on initialization, to establish configuration information, security information, and register its presence to a CDS. The COE provides a standard server initialization routine.

### **8.2.3 Standard Client Binding**

DCE provides an “automatic” binding routine that will find a suitable server and make a connection. However, this does not work for secure connections or the recommended object model. The alternative requires the client to deal with CDS querying, security, and the possibility of missing servers. The COE provides a standard client binding to allow COE clients to make a single call and not have to deal with this level of complexity.

### **8.2.4 Standard Reference Monitor**

Secure DCE servers must implement a Reference Monitor (RM) routine to verify the client’s credentials against a server’s access control lists (ACL), and an ACL manager to maintain application ACLs. For the DII COE, a standard RM and ACL manager are provided as a library routine to every server developer so that security decisions are made in a standard, certifiable manner. The OSF provides a boiler-plate RM, which has been parameterized and “segmented” for use by DII applications.

### **8.2.5 DCE Verification**

The `VerifySeg` tool includes verification of DCE application segments. Refer to Chapter 5 for the appropriate segment descriptor entries, and to subsection 8.3.4 below for a brief synopsis of the required segment descriptors. COE tools verify that a DCE

segment has been properly installed, and that CDS entries meet the COE guidelines and agree with the entries in the relevant DCE segment descriptor.

### **8.2.6 Template Application**

Creating DCE segments can be difficult because of complexities within DCE itself. To aid segment developers, the COE Developer's Toolkit contains an example template application. This application serves as a working model and template for developers of other DII COE applications using DCE.

## 8.3 Runtime Environment

Many of the security-related objects and concepts within the rest of the COE and Unix have counterparts within DCE, although the DCE object often has more powerful features and attributes. This section states requirements for the development of client/server applications using DCE. The guidance provided shall be followed by all DII applications using DCE, including applications that do not yet fully comply with the DII COE. Failure to comply with this DCE guidance may result in operational conflicts between applications.

This section begins with a description of the directory structure required for DCE segments. The general structure for segments is described in Chapter 5, but it is useful to collect the information into this section as an easy reference for relevant information. Then, the conventions for CDS and DFS for the COE are described. A summary of segment descriptors relevant to DCE are described and the remainder of this sections gives specific information on COE conventions for DCE, organized by server and client.

### 8.3.1 Segment Directory Structure

DII segments are delivered in accordance with a fixed file/directory structure defined in Chapter 5. Some DCE information is also delivered in Unix files. Other information, such as CDS information, must be delivered as files and built in CDS as part of installation.

Figure 8-1 illustrates the DII COE directory structure for segments. The shaded portions indicate the additional information required which is DCE-specific. Chapter 5 contains information about segment descriptors that are required for all segments, including DCE segments.

The additional information required to describe DCE segments is as follows:

- IDL for all interfaces shall be delivered in files of the form *interface.idl* in the segment's *include* directory, where *interface* is the name of the interface.
- DCE installation/deinstallation *dcecp* scripts shall be delivered in files named *dce\_install.dcp* and *dce\_deinstall.dcp* in the segment *SegDescrip* directory.
- Additional DCE-related configuration information is recorded in the *DCEDescrip* segment descriptor. See subsection 8.3.4 below.

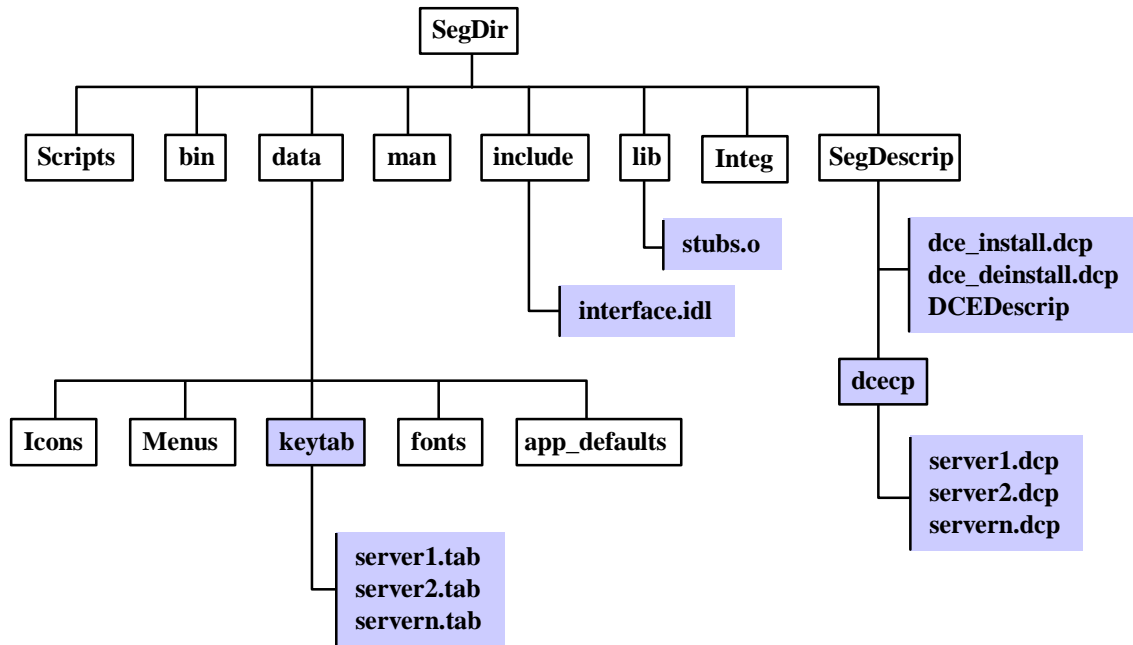


Figure 8-1: COE Directory Structure for DCE Segments

### 8.3.2 Cell Directory Service Structure

Figure 8-2 illustrates the CDS structure for a DII COE cell.<sup>3</sup> The following description summarizes the structure:

- Server configuration entries are included under

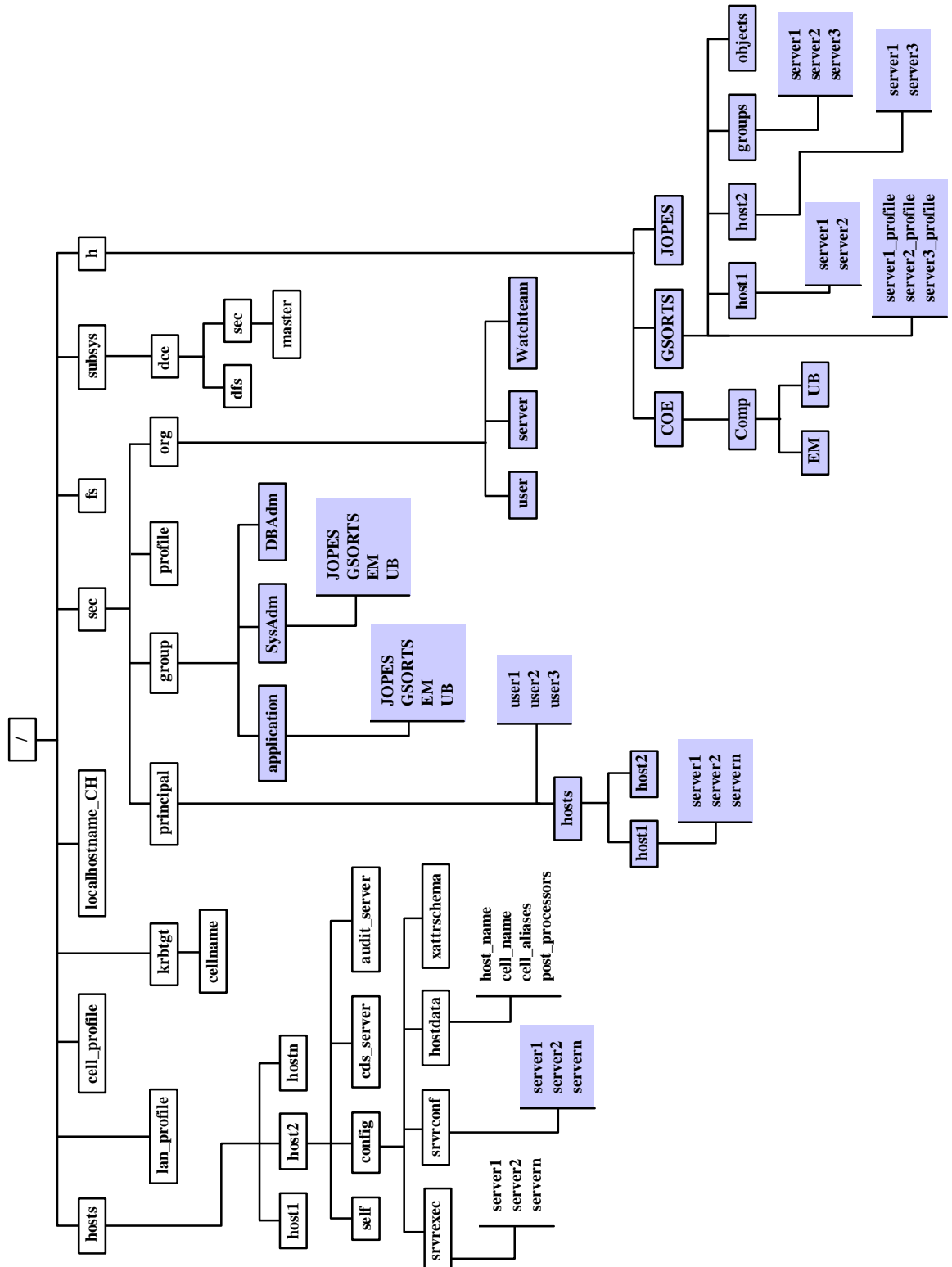
`/.:/hosts/hostname/config/srvrconf/servicename.`

These entries will be built by the segment DCE installation script.

- User principal DCE entries have the same name as the Unix userid. They are included in CDS under `/.:/sec/principal/username`, but can be referenced in security APIs using just the *username*.
- Server principal DCE entries have the name `hosts/hostname/servicename`. These entries are referenced in CDS under

`/.:/sec/principal/hosts/hostname/servicename.`

<sup>3</sup> Although the CDS directory is described using notation that is similar to the Unix directory/file system, the CDS is entirely independent from the Unix file system. The CDS structure includes *containers* that correspond with Unix directories, and *entries* that correspond to leaf nodes or files.



**Figure 8-2: CDS Layout for the DII COE**

- Security groups and organizations also appear in CDS under `/.:/sec`. (Directories `/.:/sec/group` and `/.:/sec/org` respectively.)
- All server binding entries are contained under `/.:/h`. There is one container for each mission-application segment, named with the segment's assigned directory, and one container for the COE, with sub-containers for each COE segment.
- Each segment container contains a profile entry for each service offered by the segment. This entry is named `/.:/h/SegDir/servicename_profile` and serves as the starting point for all client binding searches.
- There will be a service binding entry for each server for each host on which the server is installed. The entry has the form  
`/.:/h/SegDir/hostname/servicename.`  
The name of each entry matches the service name.
- A `groups` container under each segment is used to store any server group entries used in the binding search path.
- An `objects` container under each segment is used to store any object entries used to locate object resources used in binding searches.

### 8.3.3 Distributed File System

**Note:** The DFS global cell directory structure is still being designed. COE developers who intend to use the global cell must contact the DII COE Chief Engineer.

### 8.3.4 DCE-Related Segment Descriptors

Chapter 5 details the segment descriptor information required for DCE segments. A synopsis of the information is presented here as an aid to locating DCE-relevant information. Refer to Chapter 5 for detailed discussion.

- The `$SERVICES` keyword in the `COEServices` descriptor should not be necessary for DCE applications, since endpoints are defined dynamically.
- The `$SERVERS` keyword within the `Network` segment descriptor shall not be used for DCE services. Instead, use the `$DCESERVERS` keyword in the `DCEDescrip` segment descriptor.
- The segment descriptor `Permissions` may be used, but it is preferable to implement the application using DCE security services.

- The `DCEDescrip` segment descriptor has options `$DCEBOOT` and `$DCEDEMAND` for DCE servers started by `dced`.
- Include a `$PASSWORDS` keyword in the `COEServices` descriptor to establish a Unix userid for each server principal.
- Document DFS files used with the `$DFSFiles` keyword in the `DCEDescrip` descriptor.

This information is used to automatically configure, and verify, DCE CDS usage.

### **8.3.5 Server Issues**

This subsection deals with issues involved in the design and implementation of DCE server applications.

#### **8.3.5.1 Naming**

The following guidelines apply to the naming of servers, interfaces, CDS names, and operations:

- The service name is the name that represents the logical service provided by a server. In the non-DCE world, this name is put in the `$SERVERS` keyword. The purpose of `$SERVERS` is so that a client does not have to reference the actual hostname of a server. Examples are `masterTrk`, `slaveTrk`, `masterComms`. DCE servers are not tied to a specific host and hence do not use the `$SERVERS` keyword (Network segment descriptor). The `$DCESERVERS` keyword (`DCEDescrip` segment descriptor) is used instead to list the services offered by this segment. The `$SERVERS` and `$DCESERVERS` keywords are mutually exclusive.
- The following convention shall be used to assign service names: A segment offering a single service shall use names of the form *SegPrefix\_server* where *SegPrefix* is the segment's prefix. Segments offering multiple services shall use *SegPrefix\_service* where *service* is a meaningful name for the service. This convention will be used in naming many DCE resources associated with a service and will be represented in the text as *servicename*.
- Interface names also will be controlled to avoid duplication. The interface names shall be descriptive of the function of the interface. Each interface shall include the segment prefix. Examples are: `MAP_location`, `MAP_access`, and `MAP_rdaclif` for a segment, whose segment prefix is `MAP`, offering three interfaces. Operation names become the names of remote APIs and shall also begin with the interface prefix or a subset of it (i.e., `location_find`, `access_read`, `access_update`).

Operation names shall also be consistent with other COE requirements on naming of APIs.

DCE will automatically provide a management interface for server applications. The only management operation that is controlled is shutdown, which can only be performed by `dcled`. If a server wants to restrict other management functions, the server must deliberately disable them using the `dcled` management routines: `dcled_server_disable_if()` and `dcled_server_enable_if()`. Further information on server management can be found in Chapter 8 of the *OSF DCE Application Development Guide--Introduction and Style Guide (Rev 1.1)*.

DCE will also automatically add an interface for managing ACLs. The example interface `MAP_rdaclif` mentioned earlier uses the ACL manager API, `rdaclif`. The `rdaclif` interface consists of remote procedures called by `acl_edit` and includes remote procedures to retrieve an ACL, replace an ACL, and test whether a given client is allowed to perform a given operation.

- Names of services and interfaces need not be registered with DISA for approval. Inclusion of the segment prefix ensures that names are unique.

The CDS directory is a naming system somewhat like a filesystem. It uses a similar convention for naming its objects and directories. For example,

```
/.: /h/JOPES/JOPESdb_server
```

Servers typically use CDS for storing information about the location, interface numbers, and objects (i.e., resources) which they offer. Use of CDS naming requires as much rigor as does file system naming.

- Every DCE server segment shall be assigned a directory structure within CDS that parallels its file system location (e.g., `/.: /h/SegDir` where *SegDir* is the segment's assigned directory). All CDS entries related to this segment are contained within this directory.

In DCE, every DCE server runs under the identity of a DCE principal. Even servers offering the same service but on different machines require a unique DCE identity in order to provide reliable authentication and authorization. DCE principal names are directly tied to the CDS so server principal names can be expressed as a global name or as a cell relative name. The global name is considerably longer due to the need to unambiguously specify a principal regardless of the cell from which it originates. Within a cell, the principal can be named without including any cell identifiers because DCE will automatically append the cell information during processing.

- The convention for a DII DCE server is to use the principal name `/.: /hosts/hostname/servicename`. Each DCE principal contains

information relating to a Unix account that contains its uid. If each principal of the same service had a unique uid, control of server file system resources would be difficult. Each server providing the same service will share a Unix uid by creating principal aliases. This allows each server to have a unique account with its own password, home directory, etc., but yet share the same DCE principal and Unix account.

- There will also be a security group created for every DCE service. This group will contain all the principals that represent the servers for this service. The purpose of this group is to allow instances of a service on different machines to trust one another. The name for this group will be identical to the *servicename*. Therefore a segment containing multiple services will have multiple security groups. If an application requires additional DCE groups, they will all be prefaced with the segment prefix.

### **8.3.5.2 Interface Definition**

DCE application interfaces are defined using the DCE Interface Definition Language (IDL) defined by OSF. All interfaces are identified with a globally unique identifier that ensures that clients bind to a server offering the proper interface. IDL interfaces also allow the identification of versions of an interface. The version numbering scheme allows clients to bind to a server offering any compatible version. Assuming upward compatibility, versioning allows servers to be upgraded independently of clients, and allows old clients to continue to operate with new servers.

- DII-compliant applications shall make use of version numbers and shall provide upward compatibility between versions.

### **8.3.5.3 Server Registration**

Servers record information (bindings) in CDS that identify the interface resources and server location so that DCE clients can find the server when a client requests its service. DCE stores information in CDS structures in three types of records: *profiles*, *groups*, and *server* entries. The record name within CDS that the client accesses can correspond to a specific server, a group of servers, or a CDS profile.<sup>4</sup> Servers within a group are considered to be completely interchangeable, and are selected at random. Profiles allow the selection of alternative servers based on priorities.

Registration of DCE services shall follow the following guidelines:

- The server registration information within CDS shall follow the structure shown in Figure 8-2, which uses the mission-application segment GSORTS as an example. Each segment shall have a directory under */ . : /h* corresponding to the Unix file system

---

<sup>4</sup> The term *CDS profile* refers to a CDS entry used in locating alternative instances of a service. It has no relationship to the term *profile* used elsewhere in the *I&RTS* to identify applications and resources available to a class of users.

directory for the segment (see Figure 8-1). For example, if *SegDir* is the segment's assigned directory, it will have a CDS entry of `/.:/h/SegDir`. (The segment's assigned directory, *SegDir*, is established when the segment is registered.) Note that COE-component segments, in the Unix file system, are underneath `/h/COE/Comp` so their corresponding CDS entry is `/.:/h/COE/Comp/SegDir`. Within the segment directory, individual server instances will be registered under a directory for the host on which the server is installed. The name of the server entry shall be the *servicename*.

- A profile entry shall be created for each service directly under the segment directory using the name *servicename\_profile*. A service can also use RPC groups to collect a set of equivalent servers. Group entries shall be placed under `/.:/h/SegDir/groups`. The segment developer shall use the profile entry as the starting point for binding requests within a client application. This is the name that will be addressed by clients seeking a server.
- The server entry directly under the segment directory will always be a CDS profile entry. The name will have the form *servicename\_profile*. In the simplest case, the profile will contain a single entry, pointing to the server entry for the host on which the server is actually installed. However, by making the client address a profile entry even in this simple case, the server can be moved, or alternative servers implemented, with no changes to the client.

For example, in Figure 8-2, the GSORTS segment contains three servers: `server1`, `server2`, and `server3`. The `server1` software is installed on `host1` and `host2`, `server2` is installed only on `host1`, and `server3` is installed only on `host2`. Each server instance is registered in CDS, as shown above, during segment installation. The CDS profile entry `server1_profile` will contain pointers to the two instances of `server1`, with appropriate priorities depending on whether these are equivalent servers or one is a prime and the other a backup. The `server2_profile` and `server3_profile` entries will point to the respective server entries. Note, however, that by simply installing a new instance of `server2` and making the proper entries in CDS, a client will be able to locate alternative instances of `server2` with no application software changes.

- Servers may implement a more complex arrangement of CDS profiles and groups within this structure. A group directory will be created under the application's assigned directory as well as an `objects` directory. The naming of entries underneath `groups` and `objects` is completely under the control of the developer, within the structure above.

The DCE API supports the registration of servers at execution time by the servers. However, to reduce the volume of changes, it is recommended that DII applications build most of the structure in advance, lacking only the specific endpoint information. The specific endpoint (i.e., TCP port) is supplied at runtime to the endpoint mapper and is not

stored in CDS. Building the structure in advance also allows it to be constructed using `dcecp` rather than the more complex C-language API. Installation scripts are discussed in more detail below.

- DII-compliant applications shall register servers within CDS during segment installation. The exception to this will be for tactical applications that are installed on systems that are transient members of cells.

**Note:** This means that the CDS registration structure is not an indicator that a server exists. The client needs to actually check to make sure the server is alive.

- DII-compliant application servers shall use `rpc_ep_register()` on server startup to register the endpoint with the endpoint mapper. This call is part of `server_initialize()`, as discussed below.

The structure above is designed for the case where service is provided by servers within the local cell. However, DCE has no restriction on the location of the server. A profile entry may point to servers in a foreign cell. This allows a profile to be constructed such that, for example, it would look for a server first in the local cell, then within a near-by cell, and then anywhere. Profiles can also be used to establish preference for servers based on other criteria as well, such as the performance of the server hardware, or to allow clients to select servers with compatible data representations to reduce data conversion overhead.

The following guidance is provided on the use of cross-cell profiles:

- The required approach for accessing cross-cell services is to have a profile in each cell that references local profiles on remote cells. The starting profile has the same name of the profile that is configured into all clients. That is,

`/.:/h/SegDir/servicename_profile`

The local profiles will be similar to the profile set up for a single-cell implementation, and will point to all servers within the cell. The primary profile gives priority to servers in the local cell before looking in a foreign cell. This is illustrated in Figure 8-3. The local profile could also be a group if the local servers are equivalent. A profile is required if one server is the master and one is a backup.

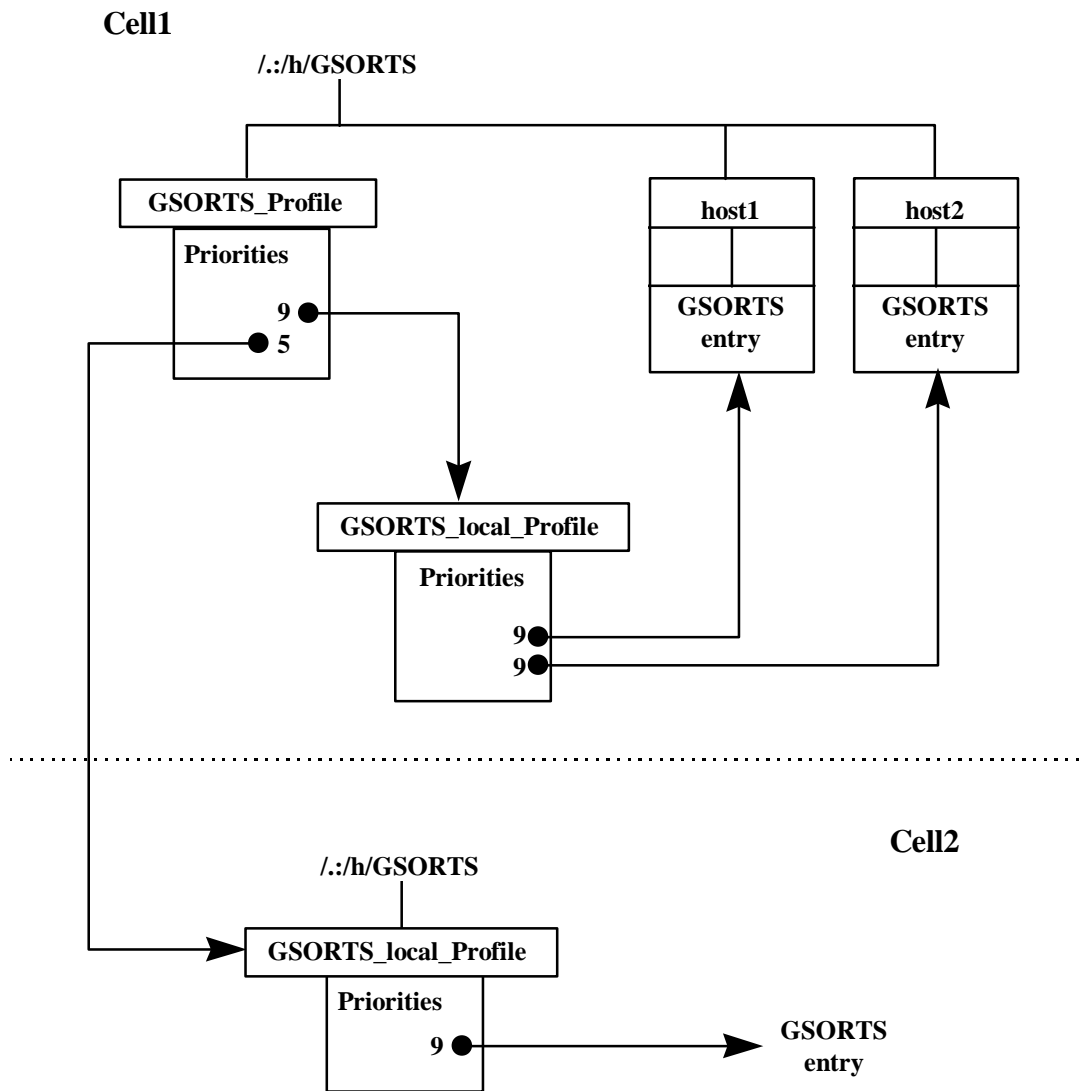


Figure 8-3: Access to Servers in Local and Foreign Cells

### 8.3.5.4 Server Startup

DCE servers are normally started by means outside of DCE's control after the DCE environment is started. DCE 1.1 introduced a facility for managing the startup and monitoring of DCE servers. This facility is provided by the `dced` daemon and facilitates full secure and remote control. When used in conjunction with the Client binding recommendations below, servers can be started only as needed, and can be restarted in case of failure, and can even be started along with any prerequisite processes as needed. The `dced` process runs as root and is the parent of all DCE servers. Using the configuration information that it stores, it can start the server under any `userid/group` pair in any directory. The `$DCEBOOT` and `$DCEDEMAND` keywords are used with the

DCEDescrip segment descriptor to identify DCE servers started by dced at boot time or on demand.

The server startup function `dce_server_register()` is provided in order to simplify the development of servers. Unfortunately, not all DCE 1.1 vendors provide this API. The function is included in the `dce_server_initialize()` API discussed below.

### **8.3.5.5 Configuration**

DCE servers contain a large number of configuration attributes that are often hard-coded in the application. The coding of these attributes makes servers hard to change or move and maintain. The `dced` daemon maintains an extensible server configuration database. DCE servers use this database to obtain their configuration information. This database is secure and is remotely manageable. When `dced` starts a server, it establishes an environment for the server based on its configuration record and allows the server to read additional initial information, similar to the windows `.ini` file.

Server configuration information is maintained in CDS under a name of the form

```
/.:/hosts/hostname/config/srvrconf/servicename.
```

For more information, refer to the *DCE Administration Guide*.

The configuration information which `dced` currently maintains is shown in Table 8-1.

The configuration information is easily extendible by teaching the `dced` about new configuration attributes. Additional attributes can be defined for any DII application as needed. Attributes will be assigned names depending upon their scope. Attributes that are required as part of COE support shall be named:

```
/.:/hosts/hostname/config/xattrschema/COE_attributename.
```

Attributes that are specific to a server segment shall be named:

```
/.:/hosts/hostname/config/xattrschema/SegPref_attributename
```

where *SegPref* is the segment's prefix.

In the case of COE-component segments, adding an attribute requires prior approval of the DII COE Chief Engineer. For mission-application segments, approval is required of the cognizant Chief Engineer.

- Application developers are responsible for creating configuration entries as part of their segment installation scripts (`dce_install.dcp` and `dce_deinstall.dcp` shown in Figure 8-1) invoked at installation time.

Information	Description	
arguments	command-line arguments required by the server	
directory	the home directory in which to start the new server	
gid	the group identity under which the server will run	
keytabs	a list of keytab object UUIDs where the server stores its keys. Although a list is permitted, only the first one is used.	
program	the name of the server program to run	
prerequisites	a list of server configuration object UUIDs which must be running	
principals	a list of server principal names under which the server runs. Although a list is permitted, only the first one is used.	
starton	a list of modifiers for starting conditions (boot, explicit, failure)	
uid	the Unix uid under which the server will be started	
uuid	a uuid which is assigned to the server object	
services	the DCE information about the operation provided. The following information is defined for each operation:	
	annotation	string describing the service
	binding(s)	protocol sequences which register the service
	flags	modifiers affecting the service's mapping {disabled}
	ifname	the interface name
	interface	the interface UUID
	objects	a list of object UUIDs associated with the service

**Table 8-1: dced Configuration Information**

- If the application is started by `dced`, the DCE daemon will ensure that the appropriate environment (e.g., Unix uid, gid, home directory, and calling parameters) is established before starting the server. The server will use the `dce_inq_server()` API to obtain its configuration record. There is no requirement for the server to use the configuration information, except to retrieve any relevant extended attribute information and pass it to the initialization routines. Servers not started by `dced` must use the `dced_object_read()` API to obtain this information.

### 8.3.5.6 Initialization

Every DCE server performs a set of functions in order to initialize. This includes registering one or more groups and entries in CDS (if not already created), and creating and registering endpoints with the endpoint mapper. In addition to these functions, a secure server must establish its identity (login), refresh its login context, and periodically change its password.

- Servers do not normally need to perform CDS registration or unregistration during server startup or cleanup. This is not necessary because the DII COE environment is rigorously defined and because a client does not use the presence of CDS information as indication of server liveness. Registration is normally performed as part of server installation.
- Servers in a tactical environment may perform registration at cell configuration time, or the first time a server initializes.

Without using a common server initialization API, a server normally performs anywhere from six to thirty API calls. (See the O'Reilly *DCE Security* book for an example of the API calls required for a secure server.) The sequence of calls is nearly identical for all servers in a well-controlled environment like DII because the parameters are defined by the configuration record.

**Note:** A common `server_initialize()` API is defined and provided as part of the COE to perform these actions. This routine initializes the server, including security, using the server's configuration information.

A server using a special initialization sequence (as defined above) can retrieve its configuration information to perform initialization. Following this guidance will allow servers to be started on demand and can be truly configuration-less.

One of the most critical initialization functions of a server is to register endpoints with the endpoint mapper in `dced`. This too is easily accomplished with the `server_initialize` API.

### 8.3.5.7 Security

To write a secure DCE application, besides the application code, the application developer needs to write client code that obtains the proper authentication and forwards it to the server. Clients are usually authenticated by the inherited login context created after `dce_login`. The COE provides a unitary login feature so that DCE login is performed as part of user login. To use authenticated RPC, a client adds a single call to the API `rpc_binding_set_auth_info()`. Clients that use automatic binding will need to use the `binding_callout` option in the ACF file.

Once the client has been authenticated, the server code gets the privileges of the calling client and determines the level of authorization possessed by the calling client. This code is called the reference monitor and it performs the authorization checks. The reference monitor receives the client access request from the server, retrieves the ACL of the object requested and checks the client's authorization against the ACL. The DCE Security Service supports two authorization protocols that can be used with authenticated RPC: DCE authorization and name-based authorization. The DCE authorization protocol is based in part on the POSIX file-protection model, but is extended with ACLs. An ACL is a list of entries that specify a privilege attribute (such as group membership) and the permissions that may be granted to principals who possess that attribute.

- To be DII-compliant, applications shall only use DCE authorization.

### **8.3.5.7.1 Authentication**

Secure servers require DCE security accounts in order to participate in DCE authentication. Each account consists of a principal, and membership in a single primary group and organization. The name of the account is identical to its principal name. DCE security names can be as simple as

`comms_server`

or hierarchical such as

`hosts/hostname/mapserver.`

- COE hosts shall use DCE principal names that align one for one with Unix operator names for interactive users. This will allow the use of the integrated login application supplied with DCE. Non-user principals associated with DII servers shall use `hosts/hostname/servicename.`

The following DCE Security Service application program interfaces can be used to perform login for a non-interactive principal:

```
sec_login_setup_identity()  
sec_key_mgmt_get_key()  
set_login_validate_identity()  
sec_key_mgmt_free_key()  
sec_login_certify_identity()  
sec_login_set_context()
```

These functions will be performed automatically when using the API, provided by DCE, `dce_server_sec_begin()`.

Secure servers must store their passwords in files since they are not capable of normal interactive login. These files are known as *keytab* files.

- For the DII COE, each application segment shall use its own keytab file. Servers shall use names that are of the form *servicename.tab*. Keytab files will be placed in the directory */h/SegDir/data/keytab* as shown in Figure 8-1. This directory must have access permissions set so that only the server principal can read or write to it.

Once a server establishes its login context, it is responsible for refreshing the context before it expires and changing passwords before they can expire. The API for managing password expiration is `sec_key_mgmt_manage_key()`. This function does not return and requires a dedicated thread.

The APIs for login refresh are:

```
sec_login_get_expiration()  
sec_login_refresh_identity()  
sec_key_mgmt_get_key()  
sec_login_validate_identity()  
sec_key_mgmt_free_key()  
sec_login_certify_identity()
```

### 8.3.5.7.2 Authenticated RPC

A client program calls `rpc_binding_set_auth_info()` to specify how an authenticated RPC connection will be set up. There are three important parameters that must be provided: authentication service, authorization service, and the protection level. Developers should use the following settings for these parameters:

**Authentication Service.** The default for DCE applications is *dce\_private*, which uses private key authentication. No other parameters are valid for DII DCE.

**Authorization Service.** An application can specify three possible values for the authorization service: *dce*, *name*, and *none*. The value '*dce*' means to pass a Privilege Attribute Certificate (PAC). This is the setting that shall be used for all DII DCE segments.

**Protection Level.** DCE allows an application to specify just how much the data in an RPC should be protected. These are: *none*, *connect*, *call*, *packet*, *integrity*, *privacy*. Integrity provides an authenticated connection between parties and ensures that messages have not been tampered with in transit. Privacy provides the highest level of protection for the RPC by encrypting the data using DES. Although the SIPRNET is encrypted using NES, the DES encrypting provides additional protection from packet snooping within a site.

- DII-compliant applications shall specify at least *integrity*. The *privacy* level should be used for particularly sensitive information.

### 8.3.5.7.3 Authorization

Once the client has been authenticated, the server must make an authorization decision. The reference monitor (RM) is the server code for retrieving the client's PAC. The information from the PAC will be used by the RM to make the authorization decision. While each server can implement its own RM, DCE packages RM code in its library. The intent is for all servers to use this same library code. This will insure that access decisions are made correctly and uniformly.

The ACL is a key part of the Authorization facility. Applications must be capable of establishing and managing ACLs. DCE provides a set of APIs for using ACL managers (`dce_acl_*`).

### 8.3.5.7.4 Generic Security Service API (GSSAPI)

DCE provides a method for using DCE security without rewriting applications to use DCE RPC. DCE contains extensions to the *IETF RFC 1508* and *1509* Generic Security Services API (GSSAPI) that will allow current applications to use DCE authentication and authorization. GSSAPI DCE extensions can be easily identified since all base GSSAPI entry points start with `gss_` while DCE GSSAPI extensions start with the prefix `gssdce_`. The most important DCE GSSAPI extension is the `gssdce_extract_cred_from_sec_context`. This call returns the Extended PAC (EPAC) which contains the security attributes of the original client and any intermediate servers. The server uses the EPAC to make its authorization decisions. For more information on the DCE Security Service and the GSSAPI, see the following:

1. The Security chapters of the *OSF DCE Application Development Guide-Core Components Volume* and the *OSF DCE Administration Guide-Core Components Volume* (DCE Security Service only).
2. The (3sec) reference pages of the *OSF DCE Application Development Reference*.
3. The (8sec and 5sec) reference pages of the *OSF DCE Command Reference*.
4. Chapter 8, *DCE Security Programming*, Wei Hu, O'Reilly & Associates, 1995.

**Note:** The DCE Security Service and GSSAPI do not currently make use of Fortezza authentication or encryption. Integration of Fortezza with DCE is under investigation.

### 8.3.5.8 Auditing

DCE provides an enhanced audit facility consisting of the audit daemon, the `dcecp` control program, and the audit logging client library. An audit daemon exists on every DCE system. Applications audit events by sending RPCs to the audit daemon on the local

system. The audit daemons write the audit records to the audit log file, which stores all the event records so that they can be reviewed later. The audit daemon also maintain event filters. Event filters are data structures that determine what events should be logged. Event filters are stored in memory and in files called event selection list (ESL) files. In order to dynamically tailor the audit process, the audit daemon exports an interface that allows the control program, `dcecp`, to change the event filters and expand the range of events that should be audited.

The final process of the audit facility is the audit-logging client library. This allows an application to send audit records to the audit daemon. When an application makes a call to the library, the library checks to see if the event should be audited. If the event filters determine it should not be audited, no RPC is sent to the audit daemon.

This represents a simplistic view of how auditing takes place in DCE. More complex actions are actually taking place including the dynamic updating of event selection lists. The most important point is that applications need only work with the audit-logging API to audit events.

- DII DCE servers shall not audit to private audit files. The ‘central trail’ shall be used for all auditing.

A complete list of the DCE Audit API routines can be found in the *OSF DCE Application Development Reference, Volume 2*.

An *event* is any action that takes place and is associated with a code point in the application server code. Each event has a symbolic name as well as a 32-bit number assigned to it. Each event number is a tuple made up of a *set-id* and the *event-id*. The *set-id* corresponds to a set of event numbers and is assigned by OSF to an organization. The organization manages the issuance of the event ID numbers to generate an event number. The structure and administration of event numbers can be likened to the structure and administration of IP addresses.

The concept of events allows each DCE implementation to establish audit events for a wide variety of actions that may take place within applications. DCE has established a hierarchy of formats for events. Once again, these are similar to the class structure within the administration of IP addresses. As part of the DCE implementation, DISA will request the assignment of a Format B event number. Format B is designed to be used by intermediate-sized organizations that need the 8 to 16 bits for the event-id. This will provide for the greatest flexibility and growth. Events may also be logically grouped together into an event class. This is a case where it may be more efficient to refer to several events as a single entity/class. Event classes are assigned event class numbers by the OSF. If required, event class number will be requested from the OSF.

### 8.3.5.9 Threads

DCE automatically implements threads for server applications. The use of threads can be beneficial to allow the server to service multiple clients concurrently. The number of active threads can be controlled by `max_calls_exec` in `rpc_server_listen()`, which can be set to zero if the server software is not “thread safe.”

While the use of threads is beneficial and recommended, the following cautions are provided:

- It is well known that threads can conflict with Ada tasking. Use threads with caution with Ada servers.
- Many COTS packages are also not “thread-safe.” Calls to databases, windowing systems, and other routines should be done with caution from within a thread.
- Handling of `fork/exec` and signals is different when threads are used.

When using exceptions with threads, the application must explicitly include the `dce/pthread_exc.h` header file.

### 8.3.5.10 Installation

In addition to installing software and data to system disk, server installation must also establish entries in DCE CDS as discussed earlier.

- Application segment developers shall include `dcecp` installation/deinstall scripts in the segment descriptor directory. The installation script will build the registration structure in CDS for each interface as part of server installation. The scripts are named `dce_install.dcp` and `dce_deinstall.dcp`. These scripts must contain conditional statements to ensure that some of the entries, such as the *SegDir* container under `/./h`, are only created once for each cell. These scripts are executed automatically by the segment installer tool during segment install/removal.
- It is recommended that there be a separate `servicename.dcp` script for each interface, to simplify configuration and maintenance of server installation procedures. The primary `dce_install.dcp` script must invoke each of the individual service scripts.
- DCE installation is normally performed by the root user logged in using the DCE `cell_admin` identity. In order to reduce the exposure during installation, DCE applications will be installed in a two-step process. During the first step, the minimal set of secure operations is performed. These include:

1. Creating a DCE account using the principal *segments/SegDir*.
2. Creating a CDS directory */.: /h/SegDir*.
3. Setting the ACL for */.: /h/SegDir* to permit all functions for the principal *segments/SegDir*.
4. Creating a security group *group/segments/SegDir*.
5. Setting the ACL for the security directory *hosts/hostname* to allow the *segments/SegDir* to create principals below it.
6. Allowing *segments/SegDir* to create one account for each service implemented by the segment (object creation quota).

**Note:** This first installation step is available as a standard utility in the DII COE. It is be parameterized based on a set of DCE-related descriptors.

The second phase of DCE installation, is performed by the segment-provided scripts (*dce\_install.dcp*, etc.) and is run using the account *segments/SegDir*. It must complete the installation process by performing the following for each service:

1. Create a DCE principal (once per cell), usually with the same name as the *hosts/hostname/servicename* to be used by the server.
2. Create a binding profile for each service of the form  
  
*/.: /h/SegDir/servicename\_profile*  
  
(once per cell) and add each server entry.
3. Create a server leaf entry (once per instance)  
*/.: /h/SegDir/hostname/servername.*
4. Create server configuration entries (for each instance).
5. Create default ACLs for any server defined objects.
6. Create security entries for the segment under *application* and *group*.

**Note:** The entire installation process is automated based on information in the segment descriptor files.

### 8.3.5.11 Server Exceptions

A DCE server must have proper cleanup code. Cleanup code is responsible for graceful shutdown and includes unregistering with the runtime, removing the endpoint from the endpoint mapper, and killing any security management threads.

- Servers wishing to honor a remote ‘stop’ request, must register an authorization function using `rpc_mgmt_set_authorization_fn()`. This can be used to control other management interfaces.
- Servers shall be prepared to catch signals and perform the necessary shutdown. This can be performed by converting signals to thread cancellation and using a cleanup function (`pthread_cleanup_push`) or using the exception facility to catch the `pthread_cancel_e` condition.

```
comm_status, fault_status op();      /* in ACF file */
error_status_t op ( args ... );      /* in IDL file */
```

Alternatively, routines can return status by using the return code as follows:

```
op([comm_status, fault_status] st) /* in ACF file */
```

- All DII-compliant applications shall catch the `SIGHUP` and `SIGTERM` signals and perform a graceful termination. By convention, `SIGHUP` means to terminate as soon as practical, and `SIGTERM` means to terminate immediately.

**Note:** The initialization API is accompanied by a server termination function so that every programmer does not need to write one.

### 8.3.5.12 Client-Side Libraries

When a server is being implemented as a reusable service, it is often desirable to develop a client-side library of interface routines to isolate the client from the DCE interfaces. This is the model most often used in commercial packages that provide a callable service. The client deals only with a well-defined call-level interface, independent of the fact that operations are performed by a server. This also allows some library procedures to be performed entirely at the client when there is no need to interact with the server.

- COE services may provide an API library separate from the IDL when that will improve the efficiency or usability of the software. When a library is provided, it shall be delivered in the segment’s `lib` directory. Unless authorized by the DII COE Chief Engineer, the library must be provided for all supported COE hardware platforms.

### 8.3.6 Client Issues

This section provides guidance for client application developers to make use of DCE servers to access DCE servers.

#### 8.3.6.1 Binding

*Binding* is the term DCE uses to refer to a client locating an appropriate server prior to performing an RPC. This is another area where a DCE application writer has plenty of latitude. Binding encompasses issues such as selection of transport protocol, selecting one or multiple servers based on load, location, or other criteria. Ideally, the binding will be resilient and deal with server's dying, stale entries in CDS or endpoint maps, automated remote server startup, and meeting server prerequisites. DCE also supports three methods for binding which impact the way applications are developed (automatic, explicit, implicit).

- It is recommended that applications use the explicit binding method since it is the most flexible. In cases where preserving the API does not permit the use of automatic binding for the client, this does not preclude server's use of explicit binding. Servers should always use explicit binding so they can obtain client identity and/or client objects.
- One precaution using explicit binding is that the client is responsible for obtaining another binding should the initial handle fail (i.e. the first server is unavailable). This feature is provided automatically by the runtime when `automatic_binding` is used.
- Automatic binding does not naturally allow for secure binding or for passing an object reference for use in object binding. When using automatic binding, use the `binding_callout` ACF attribute to annotate the binding for security or object purposes. This will register a call-back routine, to be supplied by the client, that can fill in security and object information. Refer to the *OSF DCE Developers Guide - Core Components*.

**Note:** The DII COE provides a standard API that clients can use to obtain a binding handle. This simplifies writing client applications and permits the features described above to be implemented as needed.

There are two different binding models available within DCE. In the *service* model, any implementation of a service is assumed to be able to handle any request. This is appropriate for general purpose services such as math routines. The alternative is the *resource* or *object* model, in which servers also identify specific objects for which service is provided. Clients then identify both a service and an object, and DCE will bind to a server that satisfies both requirements. For example, an OPLAN database could identify the OPLANs that it contains, or a map server could identify the maps it can provide. A

client could then request “Connect me to a map server that has a map of Bosnia.” Different objects could also be used to distinguish between test and “live” versions of a database. The object model can also be used to identify a “role” being supported by a server. For example, the client could request “Connect me to a server that is supporting the ‘observer’ role.” The object model is a little more complex, but provides much greater capability.

- DII COE client/server applications should use the resource model for binding. For the simple case where there is currently no distinction among implementations, each server should register an object corresponding to the server, and the clients should request this object. This establishes the structure for greater flexibility later. It also establishes an object-oriented flavor to interfaces that may ease transition to the use of object request broker technology in the future.
- DII COE client applications need some means of learning these object UUIDs. There are two choices: define the object UUID values in ‘header’ files, or use CDS as an object catalog. Object entries in CDS shall be placed under the `/.:/h/SegDir/objects` directory or under another subdirectory under `objects` (i.e., `objects/Maps`). Groups can be used to collect these objects (for example, `groups/Maps` may refer to object entries `objects/Bosnia` and `objects/Iraq`).

### 8.3.6.2 Exceptions

*Exceptions* are a means of handling failure conditions which occur during program execution. DCE implements exceptions locally and remotely as a result of an exception occurring during execution on a server. Using exceptions requires the use of a potentially new programming style. DCE uses exceptions internally as a means of conveying the failure status of RPC communications-related failures. The default handling of an exception is a program abort which is not desirable. The choices for an application developer are as follows:

1. Use exceptions by including `dce/pthread_exc.h` and defining TRY/ENTRY blocks around code that may raise an exception.
2. Attempt to avoid exceptions by using the `comm_status` and `fault_status` attributes in an ACF file. To this end, new RPC operations should reserve use of the last parameter in each RPC as a means of conveying error status by doing the following:

```
void op ( args..., error_status_t *st);    /* in IDL file */
```

- DII applications shall make provisions for handling exceptions using one or the other of these methods. The latter method is recommended because of its language independence, but either method is acceptable.

### 8.3.6.3 Security

In DCE, the client is responsible for selecting the security protocol and level, whereas the server maintains the choice of accepting the client's request or rejecting it. The API `rpc_binding_set_auth_info()` is used to specify the client selections. The default protection level is `rpc_c_protect_level_default`. The default authentication service is `rpc_c_authn_default`. The default authorization service is `rpc_c_authz_dce`.

- DII COE clients shall use the DCE authorization protocol along with packet integrity. Applications requiring additional security should justify and identify those requirements appropriately.

In order for a client to initiate a secure transaction with a server, the client must know the server's principal name. This information along with the security level is placed in the binding handle. In the absence of a standard binding interface, the client can obtain the server's principal name using `rpc_mgmt_inq_server_princ_name` or can query the configuration record on the host whose binding was obtained from CDS.

**Note:** The latter is performed automatically by the COE supplied binding API.

### 8.3.6.4 Auditing

There is no difference between auditing in a client and in a server. However, auditing is almost always performed in a server rather than in a client. Auditing can be performed by non-DCE applications, but the user or application must perform a DCE login in order to obtain DCE identification information that is inserted in the audit records. See subsection 8.3.5.8 for a discussion of auditing.

### 8.3.6.5 Threads

While threads are not automatically enabled for DCE clients, the DCE *pthread*s package is available for use by DCE clients. The cautions mentioned under server issues apply to clients. Client application developers should read more about the implications before using threads, particularly with Ada applications. Vendor release notes should be consulted when using threads. Vendors may require the use of special compile flags such as `-D_REENTRANT` or `_THREAD_SAFE` and may need to be linked with vendor-specific libraries.

### 8.3.7 Miscellaneous Information and Requirements

This final subsection provides some remaining details for properly using DCE within the context of the DII COE.

- The COE establishes the CELL environment variable to contain the current cell name.
- Unix userid shall agree one-for-one with DCE principals.
- Each Unix group used with a DCE application shall have a matching DCE group, but not all DCE groups must have a matching Unix group.
- Account groups do not have a useful analog in DCE, although organizations or groups could fill this function.
- Unix file permissions are similar to DCE ACLs, although ACLs are much more flexible.

## 8.4 Distributed File System

DFS offers some unique characteristics as a remote file service product. Some of these capabilities are often replicated by individual applications. Using DFS can provide significant benefits to applications that need to provide coherent file access to a very large community. Using DFS, all sites have access to a single logical file space. In GCCS 3.0 this access is provided by a NFS-to-DFS gateway machine located at each of the GCCS sites. DFS also provides a built-in replication mechanism that can be used to provide rapid file access and high availability. It is fully integrated within DCE and uses secure DCE-RPC as well as DCE's fine-grained access control mechanisms.

**Note:** This section uses GCCS as an example and the guidance given is specific to the GCCS global cell. However it is also of interest to other DII developers since the techniques applied to GCCS could also be implemented for other areas.

The DFS provides a transparent, secure global file system. DFS has enormous potential for sharing files within and among sites. DFS will be installed within a global cell that has machines at four sites world-wide (DISA, TRANSCOM, EUCOM, and PACOM). This cell will provide secure, global visibility to current information using automatic replication. All GCCS sites will share files by access to a file server within this cell. Initially, DFS will be used for a limited number of files, but the usage will grow as experience is gained.

DFS provides the following features:

1. *Client-side caching:* DFS is implemented as a "stateful" file service. Servers are knowledgeable about clients, files in use, and network copies. This allows clients to maintain full disk-based copies of server files to achieve near local disk performance. This is accomplished using a token passing scheme. The NFS-to-DFS gateway machines will be configured with large disk caches (dedicated storage) for caching of remote files. The probability of finding cached data within each site, or at least within the theater, will be high and the dependency on the network is similarly reduced.
2. *Transparency (POSIX semantics):* DFS supports nearly complete POSIX semantics for file system access. This guarantees consistency of file access to non-replicated files across all DFS clients. For files that are not replicated, DFS will ensure that any file changes are immediately visible to other users of the file. Other systems with stateless implementations have far weaker semantics due to the possibility of having multiple copies in client buffers.
3. *Replication:* DFS divides file systems into smaller hierarchies called *filesets*. DFS can create replicated read-only filesets of a given master writeable copy. Replication provides load balancing and additional availability. A flexible scheme exists for keeping the master and read-only copies in synchronization within selectable time intervals. All accesses to the writeable fileset see any changes immediately, while accesses to a read-

only replica see the change after some delay, usually about 30 minutes. These consistency controls allow a trade-off between performance and coherency. In general, replication is only used for files that change infrequently.

Note that “immediately visible” is from the perspective of the NFS-to-DFS gateway. Because clients access the gateway using NFS, the NFS consistency semantics apply, and updates may not be immediately seen by the clients.

4. *Backup filesets (cloning)*: DFS provides the ability to create a backup of a fileset, and to make this backup available online as a read-only copy. The backup is accomplished using an efficient system of file pointers, so that only files changed after the backup take up additional space in the file system. The use of backup can allow users to recover overwritten or deleted files without administrative help, without doubling file space requirements.
5. *Use of DCE security*: DFS uses DCE security to provide authenticated access and ACLs for granular access. DFS ACLs are based on DCE ACLs, but implement a specific security model that is much more flexible than Unix file permission bits. ACLs can specify the access privileges afforded to specific users, any local user, users in specific named security groups, users from a specific cell, users from any external cell, any authenticated user, and non-authenticated users.
6. *Initial ACLs*: In addition to specifying ACLs for files and directories, DCE also allows a separate set of “Initial ACLs” to be attached to a directory. These specify the ACLs that will be applied to any new file created within the directory. In addition, “Initial Container ACLs” can be specified to identify the ACLs for any new directories. Among other things, these can be used to allow users to create new files and directories without allowing them to subvert the ACLs on the directory (e.g., granting public access to files in a sensitive directory).
7. *Delegation*: DFS also supports delegation of DCE credentials, which can be used to protect not only who can access a file, but also specify the means of access. For example, ACLs can permit user john to access the GEOLOC file through the GEOLOC server but prevent john from accessing the file without using the server, and can prevent another user from accessing the file even if they use the GEOLOC server.
8. *Administration*: DFS supports advanced administrative functions such as hot backup, moving live filesets between machines, quota controls, transactional file system, dynamic re-sizing of file systems and the ability to control groups of files in filesets rather than in file system units.
9. *Location independence/consistency of naming*: All DFS files are accessed by consistent names that do not contain any location information. For GCCS, a file could be in any of the global cell file servers, or replicated in multiple servers. Although

GCCS will use a single DFS cell, in general DFS uses CDS to access file systems that can easily span cell boundaries. Every client system has the same file system view regardless of the cell to which they belong.

10. *Wide-area access*: DFS is built on top of DCE RPC that can use TCP, UDP or other protocols. Because of its efficiency, circuits of 56Kbps are adequate to provide wide-area access to DFS servers.

### **8.4.1 DFS Structure**

In general, the DFS file system is a hierarchical structure starting at the `/... CDS` directory. Files in any cell can be addressed just by referencing the DFS filename. The structure of a DFS filename is `/.../cellname/fs/filesystem`. An example of a system's DFS directory is `/.../gccs.smil.mil/fs/usr/JOPEs`. The logical naming of files does not require that the files reside in a specific server. The physical representation may have files in another location or perhaps replicated across several file servers. As a convenience, a symbolic link `:/` is made to represent the files within the current cell.

**Note:** In GCCS 3.0, it is anticipated that there will only be a single global cell containing the DFS file space.

One of the primary purposes of DFS is controlled sharing of information. In the C3I environment, information sharing occurs in at least three different dimensions: within an organizational structure (e.g., across a single service or agency); within the unified command structure (e.g., among a CINC, JTF, and supporting commands); and within functional groups (e.g., among operations watchteams at all sites). All of these can be done using DCE security groups. Group ACLs may be attached to any file within a file structure, but it is most easily understood and administered if the sharing requirements are explicit in the structure. For the GCCS DFS, the file system is organized around these sharing dimensions.

### **8.4.2 DFS Guidance**

DFS should be used for files that meet one of more of the following criteria:

1. Files that are read-mostly (i.e., are read many more times than they are written).
  2. Files that require high availability.
- For files that change frequently, there is a tradeoff between currency and the overhead of replication. Changes to non-replicated files are visible immediately, while changes to a replicated file may not be visible for a period of time. The replication update rate can be set by fileset, but a long interval between replication can increase the chances of accessing a stale copy.

- Files that are site-specific must be placed in site-specific directories in DFS. Be cautious when mapping an application data directory into a shared data directory if the application has any hard-coded file names. It is possible for one site to write the file and unintentionally change the values for all sites.
- For GCCS, DFS files will initially be mapped into the local NFS file system on /GCCS. All client machines will mount /... from the NFS-to-DFS gateway machine. /GCCS will be a symbolic link to /.../gccs.mil/fs.
- If application-specific directories are used in DFS, the segment installation procedures shall create the directories. Note that the full directory names are site-specific.
- Use symbolic links to map DFS files or directories into the proper place in the local file system. All mapping shall be done at a directory level. System developers are also responsible for constructing symbolic links from the local file system to the global DFS in their installation procedures.
- Do not create a symbolic link from /.: /gccs.smil.mil/fs/ to /:/, and do not use the notation /:/ within DFS references. This notation refers to the DFS within the current cell. Since all GCCS applications operate outside the global cell, this would create an ambiguous reference if the site implements DFS internally in the future.
- Do not place RDBMS databases into DFS. The DFS file consistency and caching methods do not support the level of sharing required by and RDBMS. It is possible to back up databases into DFS for re-loading somewhere else.
- GCCS application servers, or even clients, may become DFS clients and access the global cell directly. Bypassing the NFS-to-DFS gateway may result in better performance due to local caching and better consistency semantics through avoiding NFS.

### 8.4.3 Potential Uses for DFS

Global DFS cells can be used in a variety of ways to assist operators and developers, including the following:

1. *Data distribution:* Many sites are using `ftp` as a means of obtaining remote files. The transparency of NFS or DFS is much more powerful than `ftp`. NFS is not well suited for wide-area access and has serious security issues when used across sites. The originator can simply write the data into DFS using any software, and the user can immediately read it using the appropriate application. If the originator changes the file, the other users can almost immediately see the change.

2. *Reference files:* Applications frequently use reference files for maintaining information such as maps, inventory, or flat-file databases. These files are updated by a few sites and are made available to other sites using primitive distribution techniques. DFS also has the ability to use 'cloning' whereby a virtual copy of a file is kept, but with a fraction of the storage costs. Using this feature, the global file system could make available old and new copies trivially.
3. *Secure files:* Files containing security sensitive information should not be kept in NFS file systems. DFS is a secure alternative to NFS. Using DFS, files can be distributed and controlled at whatever degree is necessary.
4. *Mobile Personnel:* Operators who travel regularly to remote sites are probably using non-secure means (i.e., telnet) to access files such as e-mail, data files (phone lists) or documents. Both telnet and ftp can provide access control, but in both cases the user's password is sent unencrypted across the network. DCE provides more flexible security and the password is never exposed on the network. By storing these files in DFS, they can be securely accessed remotely.
5. *DCE configuration information:* Information about site configuration such as its DCE configuration can easily be stored in DFS. Cell backups (critical DCE databases and configuration files) can be done remotely by writing into a global file system.

## 8.5 Migration Recommendations

Applications must be programmed to use DCE before the application can fully benefit from the power of DCE. It is assumed that the movement to DCE among applications will be gradual. Although not all applications will be re-engineered to use DCE RPCs immediately, they can still take advantage of other DCE services using techniques described in this section.

The next subsections describe four scenarios and identify ways in which DCE services can be used in each case. The example cases are not mutually exclusive in that an application may take advantage of several of them. The first two cases are specifically targeted at legacy applications, while the last two may be used by legacy or newly developed distributed applications.

### 8.5.1 Case1: Application Startup

A typical application startup scenario in the DII starts with the client workstation displaying a user desktop. The user selects an icon or menu entry, which causes a “button script” to be executed to start a DII application. The application may be local or remote. The desktop ensures that the user is authorized to select the icon or menu item. In the case of an application on a remote application server, the script uses a Unix command such as `rsh` or `rexec` to start the remote server. The server application then opens a window on the client workstation and begins a dialog with the user.

The `rsh` command requires a level of mutual trust between the application server and the client. It is possible for a malicious client to masquerade as an authorized user and run an application for which they are not authorized. This is particularly a problem for legacy applications that run under a distinguished uid, such as JOPES (i.e., not the user’s id). Use of a simple DCE wrapper can ensure the user is authorized using strong DCE protection.

Through the use of a transparent DCE *wrapper*, the startup of DII applications can be fully protected using strong DCE authentication and access controls. Instead of invoking a user application, a button-script will invoke the wrapper and pass the name of the user application and any parameters. The wrapper will verify that the user is authorized to use the application, then launch the application. The application receives control just as if the script had launched it directly, so no application changes are required. In addition to performing authentication, the wrapper can audit execution of applications.

The wrapper can be used to launch applications on the client machine or on a remote machine. In the case of a remote application, the wrapper will operate much like the Unix `rexec` or `rsh`, but will use authenticated DCE RPC to communicate to a remote wrapper server and will use the DCE ACL model. The remote wrapper will authenticate the user, verify that the user is authorized, then set up the application environment before launching the application. Unlike `rexec` or `rsh`, the button script does not need to specify the machine that contains the application. By proper use of the CDS binding

information, the wrapper can make a request such as “connect me to a wrapper server on a machine that has the JOPES application.”

The wrapper approach has the advantage of allowing full security over execution of DII applications without having to make changes to any applications.

- This temporary approach is permissible only as an interim step for legacy applications as they migrate to DCE. New distributed applications shall be designed as two and three-tier client/server applications making use of RPC. New COE-component segments shall not use this approach without prior approval of the DII COE Chief Engineer. Mission-application developers shall not use this approach without prior approval from the cognizant Chief Engineer.

### **8.5.2 Case 2: Socket/ONC RPC**

Some applications are distributed and use sockets or unsecured ONC RPC to exchange control and data. Some socket applications perform highly sensitive operations, but essentially accept any request presented to the designated endpoint. Even without converting to full DCE RPC, these applications can make use of strong DCE authentication and access control. Socket-based communication is also susceptible to packet insertion attacks.

Existing applications that use sockets or RPC and desire greater security should seriously consider migrating to use of DCE RPC. In many cases the effort to convert to authenticated DCE RPC is not great. However, even if only limited application changes can be made, the use of DCE security is possible using the new GSSAPI. With the GSSAPI, the client application obtains a user credential, which is passed to the server application. The server verifies the user credential through another call to the GSSAPI.

The simplest use of the GSSAPI will get the credential once and pass it only in the first message. This provides some measure of security, but not as much as passing the credential in every interchange. However the latter requires more widespread changes to the application. It also requires the application to periodically refresh the credential before it expires.

The following sequence of calls illustrates the use of GSSAPI:

1. Client calls `gss_init_sec_context` to obtain a security token to pass to the server.
2. Client passes token to the server across the revised socket or RPC.
3. Server receives token and calls `gss_accept_sec_context` to decode the token, then gets a copy of the session key.

If the credential is valid, the server can convert the token (session key) to a DCE client/server, which is used as the subject in the access control decision; otherwise, it rejects the request. The use of GSSAPI is discussed further in subsection 8.3.5.7, Security.

- This temporary approach is permissible only as an interim step for legacy applications as they migrate to DCE. New COE-component segments shall not use this approach without prior approval of the DII COE Chief Engineer. Mission-application developers shall not use this approach without prior approval from the cognizant Chief Engineer.

### **8.5.3 Case 3: Distributed Databases**

Perhaps the greatest potential use of distributed computing in the DII is for distributed databases, using products such as Oracle SQL\*NET. This provides some security, but requires duplicate identification of people and resources, increasing administration. It is possible to integrate database security and remote access control with DCE security using COTS.

At least two COTS alternatives have potential for providing DCE security to remote database connections currently using Oracle SQL\*NET. The first is to use the SQL\*NET DCE product as provided by Oracle. This product provides an Oracle integration of CDS and Security into existing applications and servers. The Oracle database uses the client's DCE credentials for access decisions, alleviating the need for a separate Oracle login. The product also maps DCE groups to database roles, unifying another aspect of security. The ability to map a DCE security group membership into an Oracle role will not be available until the next release. Database servers register in CDS and clients use CDS to locate a database server. Unfortunately, this product is not currently available for all COE platforms.

A second approach is to use Open Horizon's *Connection* product as a means of integrating existing Oracle database clients and servers. It uses essentially the same approach as SQL\*NET DCE, and product availability is immediate. It supports applications using OCI. In addition, this product supports the *de facto* standard ODBC remote database connection protocol, allowing access to a large number of other databases and products. Its major disadvantage is that it cannot provide DCE group to Oracle role mapping. It requires that privileged database access be granted to the *Connection* server. It cannot currently be used with applications that use ProC or ProAda embedded SQL, since these use undocumented interfaces, instead of standard OCI.

**Note:** There are no facilities to directly support either approach in the DII COE. Tools such as *Connection* are under consideration for later COE releases. Developers may make use of these tools with the COE if required. This subsection is provided only to describe a potential migration approach.

### **8.5.4 Case 4: Distributed Files**

Perhaps the easiest way to use the security features of DCE is through use of DFS. For example, the GCCS Global DFS will allow the use of DCE access control, authentication, replication, and consistency controls, with little or no application impact. It reduces requirements for user-initiated FTP and polling.

DFS offers some unique characteristics as a remote file service product. Some of these capabilities are often replicated by individual applications. Using DFS would be a significant benefit to applications that need to provide coherent file access to a very large community. DFS also provides a built-in replication mechanism that can be used for software distribution. It is fully integrated within DCE and uses secure DCE-RPC as well as DCE's fine-grained access control mechanisms. GCCS will use DFS to allow all GCCS sites to have access to a single logical file space. In later versions of GCCS, this access is provided by a NFS-to-DFS gateway machine located in each of the theaters.

The DFS provides a transparent, secure global file system. DFS has enormous potential for sharing files within and between sites. DFS will be installed to support GCCS within a global cell that has machines at four sites world-wide (DISA, TRANSCOM, EUCOM, and PACOM). This cell will provide secure, global visibility to current information using automatic replication. All GCCS sites will share files by access to a file server within this cell. Initially, DFS will be used for a limited number of files, but the usage will grow as experience is gained.

- Developers planning to use DFS or anticipating a need for DFS for COE-component segments shall contact the DII COE Chief Engineer for more detailed information and guidance. Mission-application developers shall contact the cognizant Chief Engineer to ensure that such usage does not interfere with the COE, or with other COE-based systems.

**This page is intentionally blank.**